Georgia Institute of Technology

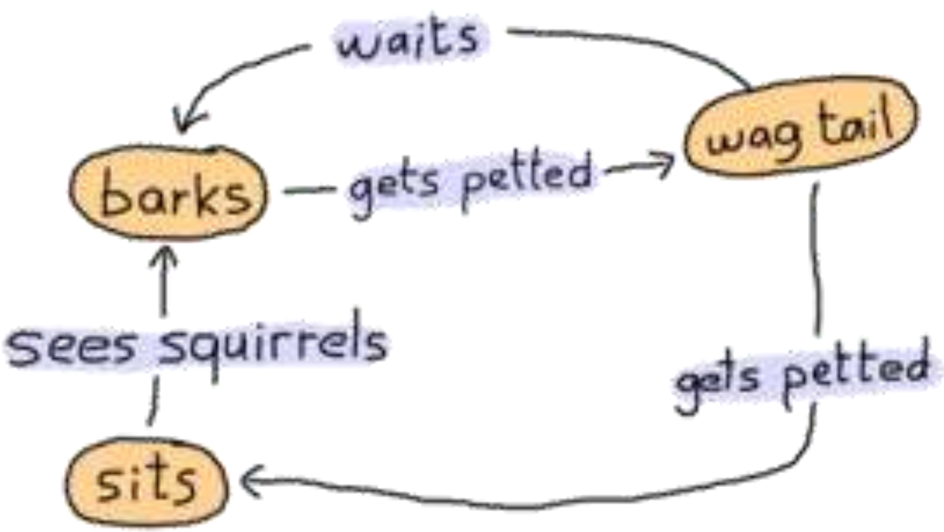
Run-time Uncertainty, Timers, and Scheduling

(Module 3)

Vertically Integrated Projects (VIP) Program

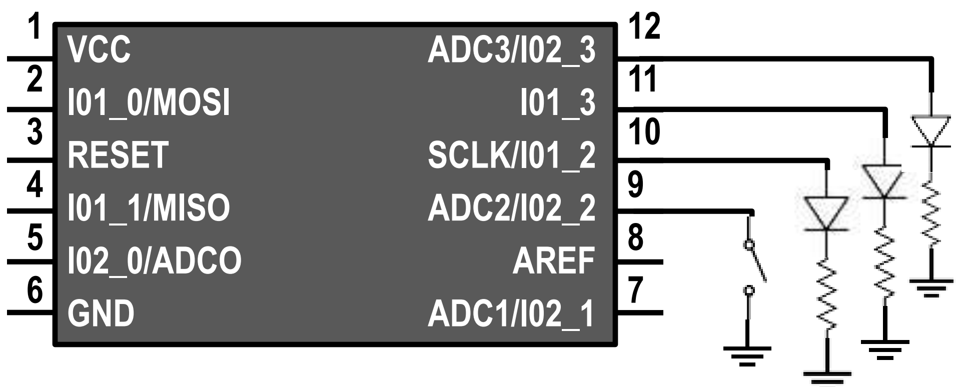
Contents

- Runtime complexity
  - Dependency injection
- Clocks and timers
  - Parts of a timer
  - Time calculations
- Activity flow
  - Scheduling
  - State machines
  - Interrupts



Runtime Complexity

- Alternating LED activation



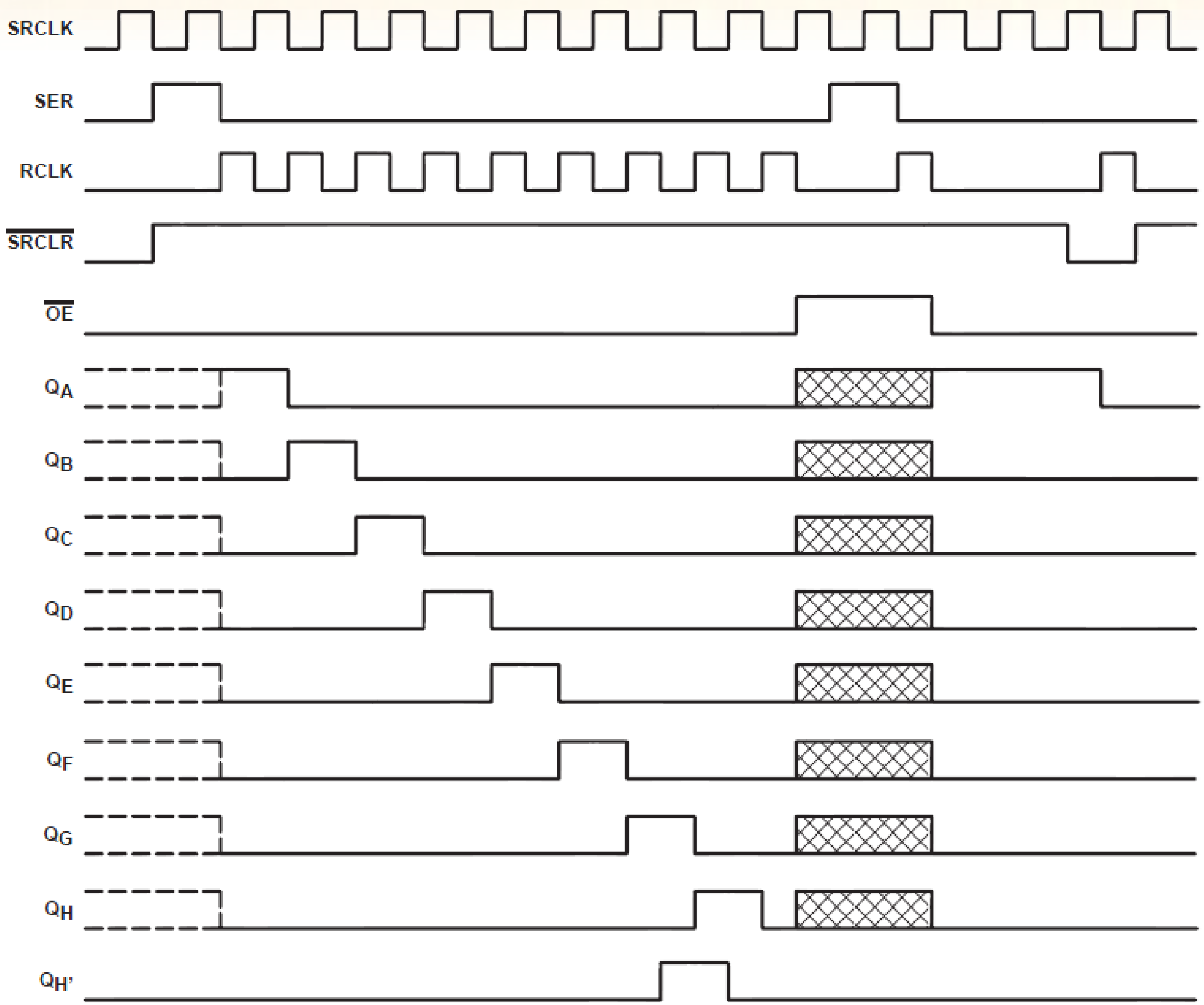
  - Makes use of RAM and saves processor cycles

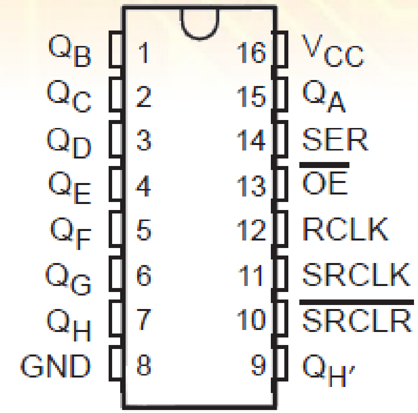
```
main loop:
  if time to read button,
    read button
    if button changed and no longer pressed
      set button changed to false
      change which LED
  if time to toggle the LED,
    toggle LED
  repeat
```

Runtime Complexity

- Dependency injection
  - Introduces flexibility beyond the use of state variables
  - Makes use of abstraction to manage dynamic changes
  - Previously, function extraction
    - LED I/O pin was hidden within the code
    - Created a hierarchy of functions with dependence on lower level only
  - Now, with dependency injection
    - Remove dependence of LED code on I/O pin
    - Passes an I/O handler as a parameter of LED initialization code
    - The I/O handler performs the changes to the LED to toglgle
    - LED will only call the I/O handler

Clocks and Timers





SN74HC595
Shift Register (8-bit)

Texas Instruments

NOTE: ☒ implies that the output is in 3-State mode.

Georgia Institute of Technology

Clocks and Timers

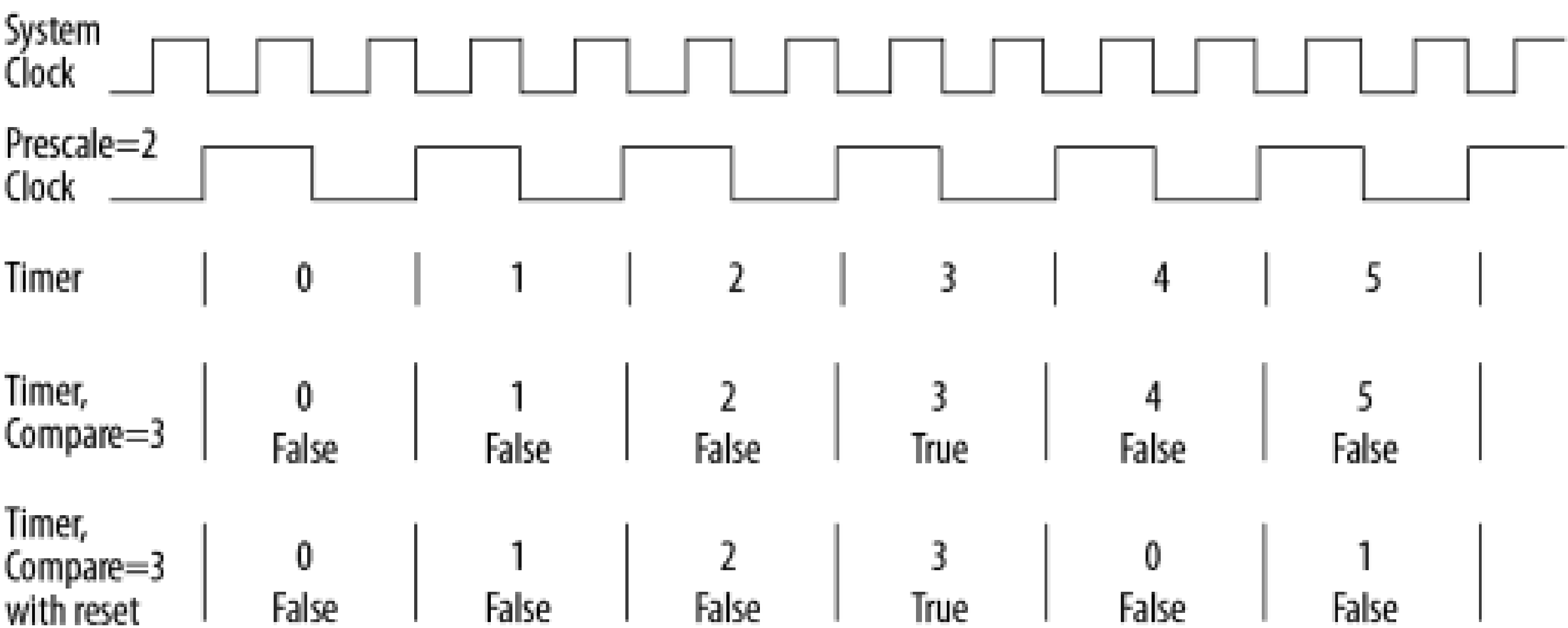
- The clock
  - Related to the cycle frequency of the processor
  - May also be related to a peripheral

- What is a timer?
  - A counter accumulating clock ticks

- Examples of maximum frequencies for processors
  - NXP LPC1313 (Cortex-M3), 32-bits, 72 MHz
  - Texas Instrument MSP430 G2201, 16 bits, 16 MHz
  - Atmel ATtiny45, 8 bits, 4 MHz

Clocks and Timers

- Timer pre-scaling and matching
  - With a factor of 2, the pre-scale clock runs at half frequency
  - Count register is set to True at 3 pre-scaled cycles
  - Upon matching, the counter may continue or reset



Clocks and Timers

- Registers that define timers

1) Timer counter
   - Contains the number of ticks since last reset
2) Compare register
   - When timer equals this register, an action is taken
3) Action register
   - Interrupt
   - Stop or continue
   - Reset the counter
   - Set an output
4) Clock configure register
   - Tells subsystems what clock to use
5) Pre-scale register
   - Sets the pre-scale factor
6) Control register
   - Starts and resets the timer
7) Interrupt registers
   - Enable
   - Clear
   - Check status

Clocks and Timers

- Time calculations

```
timerFrequency = clockIn /(prescaler * compareReg)
```

- For a ATtiny45, 8-bits, 10-bit prescale reg, at 4 MHz
  - If `compareReg` = 255
    - `timerFrequency` = 19.98 Hz; Error = 0.1%

```
prescaler = clockIn /(timerFrequency * compareReg)
          = 4 MHz / (20 Hz * 255)
          = 784.31
```

  - Another approach

```
prescaler * compareReg = clockIn / timerFrequency
                       = 4 MHz   / 20 Hz
                       = 200,000
```

    - `prescaler` can be 1000
    - `compareReg` can be 200

Georgia Institute of Technology

Clocks and Timer: Calculations

- For a ATtiny45 processor (continued)
  - Thorough timer solution

1) Minimum Prescaler

```
minPrescaler = clockIn / (goalFreq * maxCompare)
             = 4 MHz   / (20 MHz   * 255)
             = 922.72
```

2) Maximum Prescaler

```
maxPrescaler = clockIn / (goalFreq * 1)
             = 4 Mhz   / (20 Hz)          --> 1023!
```

Clocks and Timer: Calculations

- For a ATtiny45 processor (continued)

  - Thorough timer solution

    3) For each pre-scale value between 923 and 1023
       - Find a value for `compareReg`

```
compareReg = clockIn / (goalFreq * prescale)
compareReg = round(compareReg)
```

       - Find the resulting `timerFrequency`

```
timerFrequency = clockIn / (compareReg * prescale)
```

       - Find the `error`

```
error% = 100 * abs (goalFreq - timerFrequency) / goalFreq
```

    4) Find `prescale` and `compareReg`
       - With least error!

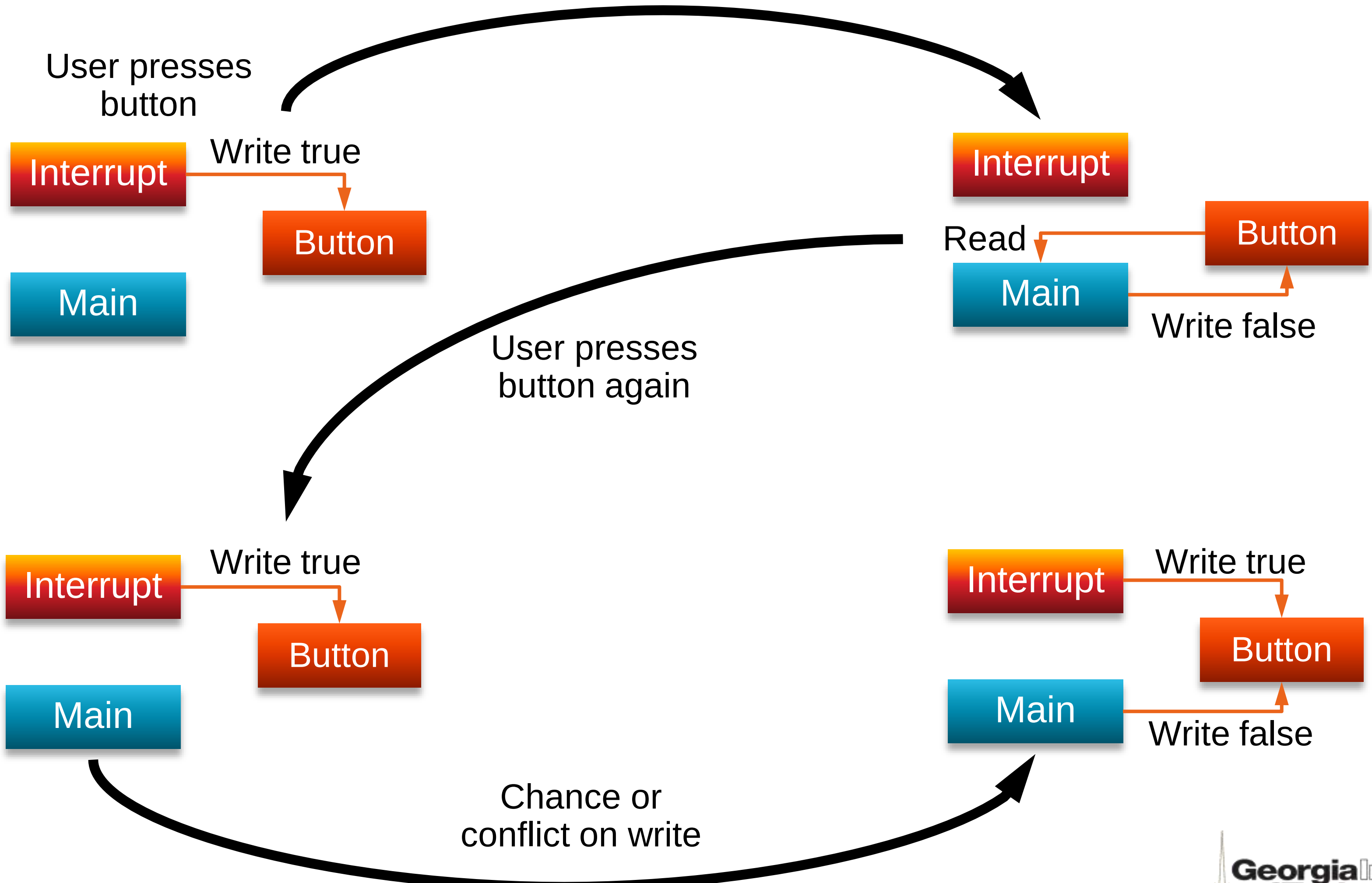
Georgia Institute of Technology

Activity Flow

- Scheduling
  - In contrast to computers, embedded systems don't necessarily use an operating system
  - Scheduling is more often set by the developer
  - Concepts to keep in mind
    - Task: is something the processor does
    - Thread: is a task plus some overhead (perhaps memory)
    - Process: a complete unit of execution with memory space
    - Mutual exclusion (*Mutex*): given a shared resource, only one task modifies it at a time
    - Atomic: referred to as an *action* that can not be interrupted by any subsystem
    - Race condition: is the uncertainty of having two modules of code setting a state at the same time

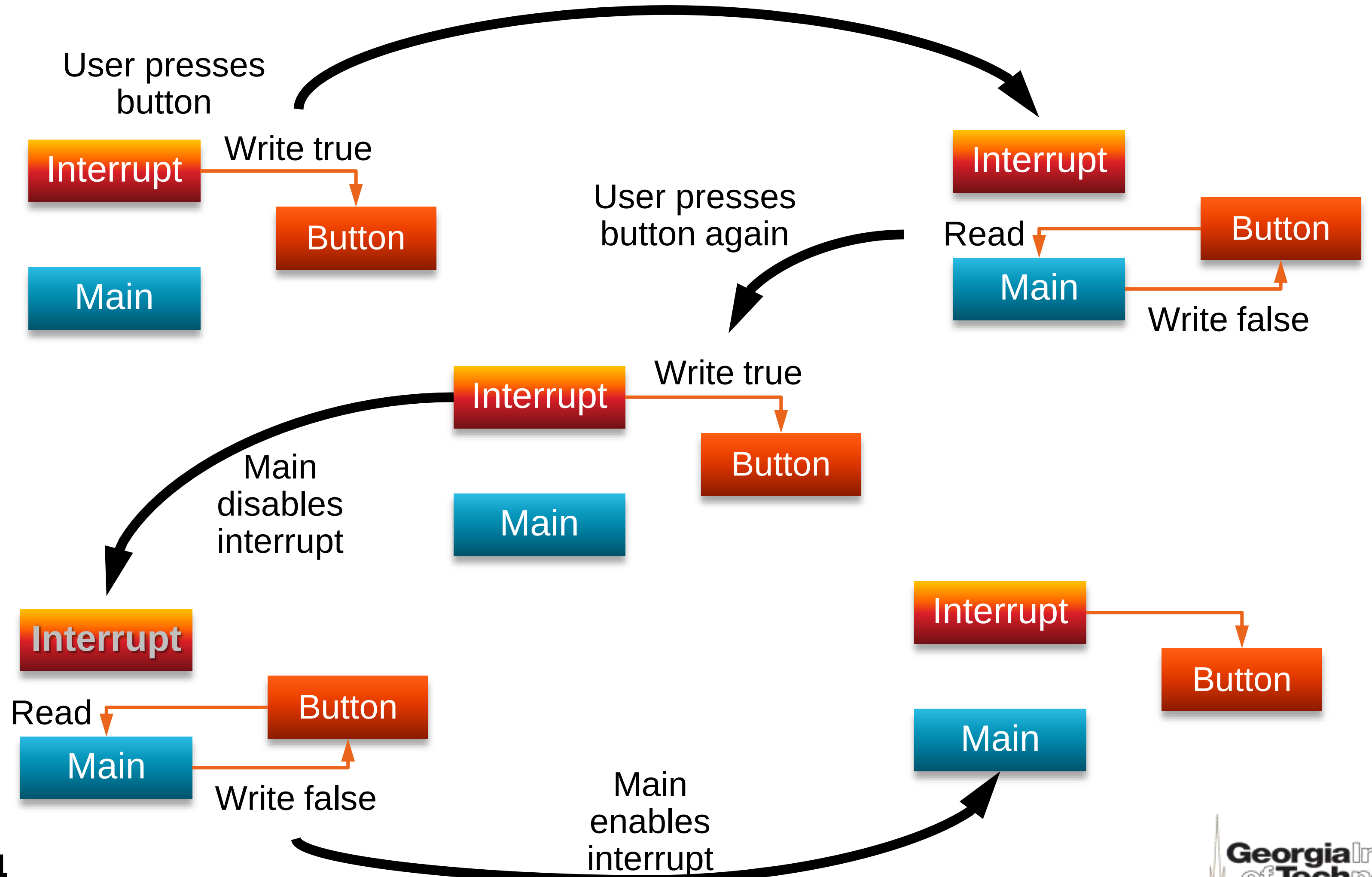
Activity Flow

- Communication between tasks
  - Race condition illustration



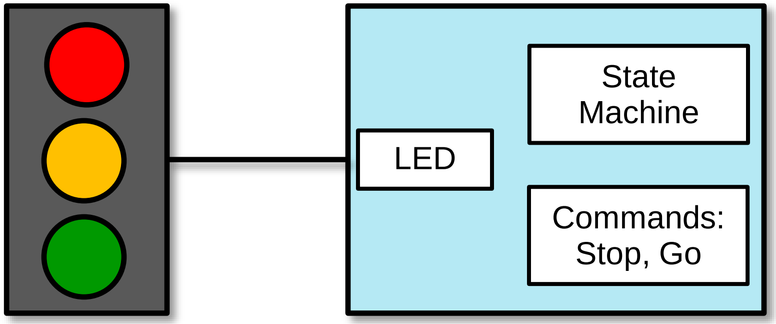
Activity Flow

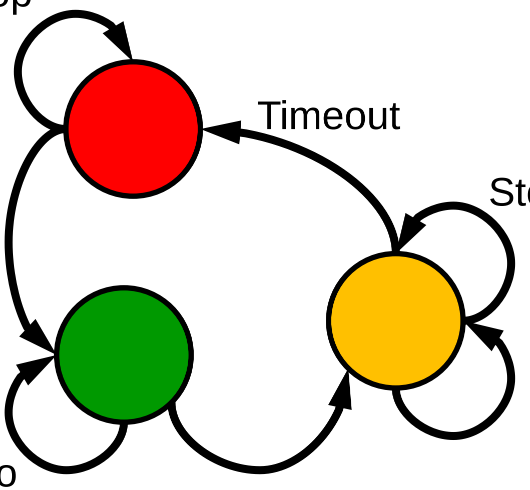
- Communication between tasks
  - Solution to an eventual race condition



Activity Flow

- State machines
  - Software pattern commonly used in embedded systems
  - Intention: “to allow an object to alter its behavior when its internal state changes. The object will appear to change its class”





Activity Flow

- State machines
  - Types
    - State-centric: seen as a big if-else statement, e.g. red, green
      - With hidden transitions: hides the “next step” in a function
    - Event-centric: based on transitions, e.g. go, stop
    - State pattern: object-oriented approach
      - Each state is an object
      - Methods in objects handle events, e.g. EventGo, EventStop
      - Higher level object (context) handles the states and calls the transitions
    - Table-driven: makes use of a table to define states and transitions
      - Easier way to document the software pattern
      - May make use of CSV and spreadsheets to handle large state machines

Activity Flow

- State machines: state-centric

```
while (1) {
  look for event
  switch(state){
  case(green light):
    if(event is stop command)
      turn off green light
      turn on yellow light
      set state to yellow light
      start timer
    break;
  case(yellow light):
    if(event is timeout)
      turn off yellow light
      turn on red light
      set state to red light
    break;
```

```
  case(red light):
    if(event is go command)
      turn off red light
      turn on green light
      set state to green light
    break;
  default(unhandled state)
    error!
  }
}
```

Generic form

```
case(state):
  if event valid for this state
    handle event
    prepare for new state
    set new state
```

Activity Flow

- State machines: state-centric with hidden transitions

Generic form

```
case(state):
  make sure about current state
  if event is valid at state
    call next state function
```

```
case(greenlight):
  turn on green (in any case)
  if(event is stop)
    turn off green light
    call next state function
  break;
```

```
next state function:
  switch(state){
  case(green light):
    set state to yellow light
    break;
  case(yellow light):
    set state to red light
    break;
  case(red light):
    set state to green light
    break;
```

Activity Flow

- State machines: event-centric

Generic form

```
case(event):
  if state transition for event
    go to new state
```

```
switch(event)
case(stop):
  if(state is green light)
    turn off green light
    go to next state
  break;
```

Activity Flow

- State machines: state pattern

```
class Context{
  class State Red, Yellow, Green;
  class State Current;
constructor:
  Current = Red;
  Current.Enter()
deconstructor:
  Current.Exit();
Go:
  if (Current.Go() state change)
    NexState();
Stop:
  if(Current.Stop() state change)
    NextState();
HouseKeeping
  if(Current.Housekeeping() state
change
    NextState();
```

```
NextState:
  Current.Exit();
  if (Current is Red)
    Current = Green;
  if (Current is Yellow)
    Current = Red;
  if (Current is Green)
    Current = Yellow;
  Current.Enter();
}
```

Activity Flow

- State machines: table-driven

STATES	Light	Go	Stop	Time-out
RED	red	GREEN	RED	RED
YELLOW	yellow	RED	YELLOW	RED
GREEN	green	GREEN	YELLOW	GREEN

```
struct sStateTableEntry{
  tLight light;     // all states have associated lights
  tState goEvent;   // state to enter when go event occurs
  tState stopEvent; // when stop event occurs
  tState timeoutEvent; //when timeout occurs
```

Activity Flow

- State machines: table-driven

STATES	Light	Go	Stop	Time-out
RED	red	GREEN	RED	RED
YELLOW	yellow	RED	YELLOW	RED
GREEN	green	GREEN	YELLOW	GREEN

```
struct sStateTableEntry{
  tLight light;     // all states have associated lights
  tState goEvent;   // state to enter when go event occurs
  tState stopEvent; // when stop event occurs
  tState timeoutEvent; //when timeout occurs
```

Activity Flow

- State machines: table-driven
  - Event handler

```
void HandleEventGo(struc sStateTableEntry *currentState){
  //turn off the light(unless we will turn it back on)
  if(currentState->light != currentState->go.light){
    LightOff(currentState->light);
  }
  currentState = currentState->go;
  LightOn(currentState->light);
  StaterTimer();
}
```

  - The table

```
typedef enum {kRedState=0, kYellowState=1, kGreenState=2} tState;
struct sStateTableEntry stateTable[]={
  {kRedLight,    kGreenState,  kRedState,    kRedState}, // Red
  {kYellowLight, kYellowState, kYellowState, kRedState}, //Yellow
  {kGreenLight,  kGreenState,  kYellowState, kGreenState},//Green
}
```

Activity Flow

- Interrupts
  - Assist in implementing the logic of state machines
  - Need to be fast!
    - Sometimes directly making use of Assembly Language
  - Bugs are difficult to find
    - i.e. as they intervene in the flow of code, changes in the execution are hard to track
  - Events of an interrupt
    1) An interrupt request (IRQ) takes place
       - from a peripheral, in the code, by a fault
    2) The processor saves its context or state
    3) The processor finds the associated callback function within the interrupt vector table (IVT)
    4) Execution of the interrupt service routine (ISR)
    5) Restore the context

Activity Flow: Interrupts

1) The IRQ (interrupt request)

- An interrupt may have been previously configured or may be included in default settings

- Configuration
  - Disable the interrupt

```
NVIC->ICER[0] = (1 << 4); // disable timer 3 interrupt
```

  - Configure it to cause an IRQ

```
LPC_TIM3->MCR |= 0x01;     //set the interrupt to occur
LPC_TIM3->MCR &= ~(0x02); // expired timer should not reset
LPC_TIM3->MCR |= 0x04;     // expired timer should stop sequence
```

  - Enable the interrupt

```
NVIC->ISER[0] = (1 << 4)  // enable timer 3 interrupt
```

Activity Flow: Interrupts

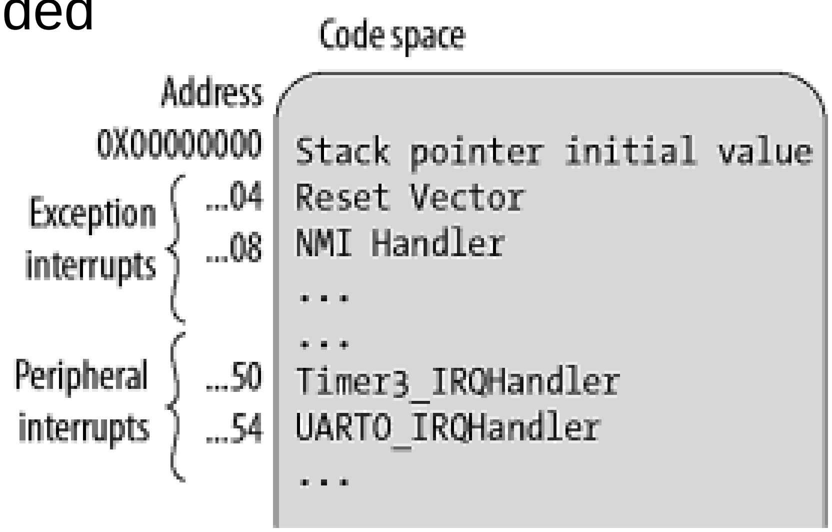
2) Saving the context

- After the IRQ, the processor saves its state in RAM
- Processor state includes
  1. Program counter (current instruction)
  2. Subset of processor registers (cached local RAM)
- Implies the presence of a latency
  - Keep a small latency!
    - Minimize the size of the context to be saved
  - Compilers sometimes force smaller latencies by limiting:
    1. the number of variables
    2. the number of nested functions

Activity Flow: Interrupts

3) Finding the ISR in the IVT

- The IVT is a list of pointers
  - Usually start-up code is available with the compiler
- A linker script sets the IVT address in memory
  - Included in the start-up code, i.e. no additional coding is needed



Elecia White, *Making Embedded Systems*, O'Reilly, 2011.

Activity Flow: Interrupts

4) Execution of the ISR: Guidelines

- Keep ISR short to manage latency
- Avoid using nonreentrant functions (e.g. printf)
  - May cause corruption of variables
- Deactivate other interrupts to avoid priority inversion
  - Macros `__disable_irq()` and `__enable_irq()` provided by vendor

5) Restore the context

- Some make use of the instruction `rti` to let know the processor the need to restore the state previous to the interrupt

Interview Question

- A small city has decided their intersection is too busy for a stop sign, and they've decided to upgrade to a light. They've asked you to write the code for the light.

  There are four lights, each with a red, yellow and green bulb. There are also four car sensors that can tell when a vehicle is stopped at the light. Where do you start? Tell me about your design, and then write some pseudo-code.

In Following Modules

- Peripherals
  - External memory
  - Buttons and key matrices
  - Sensors
  - Actuators
  - Displays
- Protocols
  - Serial
    - RS-232 and TTL
    - SPI
    - I2C
    - 1-wire
    - USB
  - Parallel
  - Ethernet and WiFi
- Integration of peripherals
- Managing resource scarcity
- Reducing power consumption (from the s/w side)